Theoretical Computer Science 132 (1994) 403–414
Elsevier

Note

On two-dimensional pattern matching by optimal parallel algorithms

Maxime Crochemore*

Institut Gaspard Monge, Université de Marne la Vallée, 2 rue de la Butte Verte, F-93160 Noisy le Grand, France

Wojciech Rytter**

Institute of Informatics, Warsaw University, ul. Banacha 2, 02-097 Warsaw, Poland

Communicated by D. Perrin
Received April 1992
Revised December 1993

Abstract

Crochemore M. and W. Rytter, On two-dimensional pattern matching by optimal parallel algorithms, Theoretical Computer Science 132 (1994) 403–414.

Simplified versions of Kedem–Landau–Palem algorithms for parallel one-dimensional and two-dimensional pattern-matching on a CRCW PRAM are presented. We show that the only nontrivial part of KLP algorithm is the preprocessing part: computation of consistent names of very small factors. The crucial part in KLP algorithm is a suffix–prefix matching subprocedure. In our algorithm such a subprocedure is avoided. A novel algorithm for 2-dimensional matching is presented which is more directly designed for two-dimensional objects. It does not use the multi-text/multi-pattern approach as in KLP algorithm. Techniques for constructing parallel image identification algorithms are introduced: cutting images into small factors, and compressing images by a parallel reduction of a large number of such independent factors into smaller objects. The importance of five types of factors is emphasized. A new useful type of two-dimensional factors is introduced: thin factors.

Correspondence to: M. Crochemore, Institut Gaspard Monge, Université de Marne la Vallée, 2 rue de la Butte Verte, F-93160 Noisy le Grand, France. Email: mac@univ-mlv.fr.

* Work by this author is partly supported by PRC Mathématiques-Informatique and by NATO Grant CRG 900293.

** Work by this author is supported by the project KBN 2-11-90-91-01.

SSDI 0304-3975(94)00012-8

1. Introduction

Two-dimensional pattern matching (2D-matching, in short) consists in finding all occurrences of an m by m pattern array PAT (filled with symbols) in a given n by n text array TEXT, where $m \leqslant n$. All our algorithms can be easily extended to the case of rectangular nonsquare patterns and texts. A position in the text array is a pair of integers in the range $[1 \ldots n]$. By an occurrence of the two-dimensional pattern PAT in TEXT we mean the left upper corner position $x=(i,j)$, where $1 \leqslant i, j \leqslant n-m+1$, of the alignment of the pattern in the text array TEXT. This means that $\mathrm{TEXT}[i+k_1-1, j+k_2-1] = \mathrm{PAT}[k_1, k_2]$ for all integers $1 \leqslant k_1, k_2 \leqslant m$. We denote by $N=n^2$ the total size of the problem.

We investigate the parallel complexity of the 2D-matching problem. Our model of computation is the CRCW PRAM (see [18] or [2] for instance). The parallel random access machine (PRAM), a parallel version of the random accessed machine, is used as a standard model for presentation of parallel algorithms. It consists of a number of processors working synchronously and communicating through a common random access memory. Each processor is a random access machine with the usual operations. The processors are indexed by consecutive natural numbers, and they synchronously execute the same central program. However, the action of a given processor depends also on its number (known to the processor). In one step, a processor can access one memory location. Parallel algorithms are presented in a similar way as in [8]. Parallelism will be expressed by the following type of parallel statement:

for all i in X in parallel do action(i)

Execution of this statement consists of:
- assigning a processor to each element of X,.
- executing in parallel by assigned processors the operations specified by action(i).

Usually the part "i in X" looks like "$1 \leqslant i \leqslant n$" if X is an interval of integers.

We are generally interested in the parallel time $T(n)$ as well as in the number of processors $P(n)$ required for the execution of the parallel algorithm. The total work done by the parallel algorithm is the product $T(n)P(n)$. There is a general fact which relates the number of processors to the total work and parallel time, see [8] for details.

Fact 1.1 (Brent's lemma). *Let A be a parallel algorithm with a computation time t. Suppose that A involves a total number of m computational operations. Then A can be implemented using p processors in $O(m/p+t)$ time.*

Brent's lemma entails an implementation problem related to the assignment of processors. There are no difficulties for applications of the theorem within the algorithms presented in this paper. In this paper, processors correspond to positions of certain texts.

Efficient parallel algorithms are those that operate in no more than polylogarithmic (a polynomial of log's of input size) time with a polynomial number of processors. The

class of problems solvable by such algorithms is denoted by NC, and we call the related algorithms NC-algorithms. A NC-algorithm is optimal iff its total work is linear. Our aim is the construction of an optimal NC-algorithm for the two-dimensional pattern matching problem.

Two fast parallel algorithms for 2D-pattern matching have been given independently in [10] and [5] (full version in [6]). We refer to them as KLP algorithm and dictionary algorithm, respectively. The first of them is optimal and the second is optimal within factor $\log(m)$. KLP algorithm uses $\log(n)$ processors less, but the dictionary algorithm is much simpler and is related to a series of algorithms for several other problems with similar complexity. The authors in [6] concentrated rather on a broad range of simple applications, than on getting an optimality result. In fact, the dictionary approach to matching problems is an old idea. It comes from [9], and it essentially appears implicitly in [2] and [1]. In [6] we stated that for fixed-size alphabets $n/\log(n)$ processors suffice for the algorithm. The proof was essentially omitted. Later, we were asked by some readers for a full proof of the assertion. This note answers such request. Moreover, it presents also a simplification as well as an alternative to KLP algorithm. The crucial part in KLP algorithm is the suffix–prefix matching subprocedure. In our present algorithm we use a more direct solution that makes the subprocedure useless. A novel algorithm for 2D-pattern matching is proposed, more directly designed for two-dimensional objects. It does not use the multi-text/multi-pattern approach as the KLP algorithm and other algorithms (see [3] or [4]) do. Four types of factors (characteristic pieces) are used: basic, regular basic, small and regular small factors. In this note, we also introduce a fifth useful type of two-dimensional factors: thin factors. Their applicability is demonstrated for 2D-pattern matching.

We assume that m and n are powers of two. This does not effect the generality of the problem since each pattern can be decomposed into a constant number of patterns whose sides are powers of two. In the case of one-dimensional patterns if the pattern size is not a power of two then we can search two subpatterns which are the prefix and suffix of the given pattern with size the largest possible power of two. In the two-dimensional case we can decompose the m by m square into 4 subsquares (not necessarily disjoint) whose sides are powers of two; we refer the reader to [6] for details. Hence, we assume later that the sizes of sides of all considered patterns are powers of two.

2. Five types of factors

The basic parts of KLP algorithm and the dictionary algorithm from [6] are constructions of dictionaries which enable to check in constant time whether some factors of text or pattern are equal. Factors of a one-dimensional string are substrings consisting of consecutive positions. In the two-dimensional case factors are rectangular parts of two-dimensional images.

Factors whose length is a power of two are called *basic factors*. In the 2-dimensional case basic factors are subsquares whose side length is a power of two. The one-dimensional basic factor f of the pattern is called a *regular basic factor* iff it starts in the pattern at a position $i \cdot \text{size}(f)+1$, for an integer i. Analogously, in the 2D case we say that the 2D factor of shape s by s is regular iff it starts (its left upper corner is) at a position such that horizontal and vertical coordinates are respectively, $i \cdot s+1$ and $j \cdot s+1$ for some integers i, j.

The total size N of a 2D image is its area. The basic (trivial) property of regular factors is given below.

Fact 2.1. *A pattern of size N has $O(N \log N)$ basic factors and only $O(N)$ regular basic factors.*

We slightly modify the algorithm presented in [6] for the computation of the dictionary of basic factors. We refer the reader to [6] for details.

We modify the dictionary algorithm (given in [6]) as follows. The modified algorithm is more optimal with respect to the work spent on patterns. It is essentially the same algorithm. The only difference is that in the pattern only regular basic factors are identified. So the total work spent on patterns is proportional to their total size (it is optimal), while for the text all basic factors are considered. The algorithm identifies all basic factors in the text. Then, we have names for all patterns and for all factors of the text which are candidates to match these patterns. There is an occurrence of the pattern at position x iff the factor starting at x and of the same shape as the pattern has the same name as the pattern. The modified algorithm is called the *partly optimal matching*.

The lemma below justifies the name of the algorithm described above. The optimality of the algorithm is with respect to the total size of patterns. The lemma is essentially already proved in [6]; it is based on Fact 2.1.

Lemma 2.2 (Key lemma). *Assume that we have t patterns, each of size M. Then, the partly optimal matching algorithm (described above) finds all patterns in a one-dimensional or a two-dimensional image of size N in time $O(\log M)$ with total work $O(N \cdot \log(M)+t \cdot M)$.*

Proof. The algorithm identifies all basic factors in the text and all regular basic factors in the pattern at the same time. The total work of this algorithm is proportional to the number of considered factors. The crucial point is that in the pattern only regular factors need to be processed. There are only $O(M)$ regular basic factors in the pattern due to Fact 2.1. Hence the work spent on a single pattern is $O(M)$. Similarly for t patterns of size M, it is $O(t \cdot M)$. This completes the proof. □

Let $k=\log(m)$. We can assume w.l.o.g. that n, m are divisible by k. The factors of length k of the string are called *small factors*. The small factors that start at positions $i \cdot k+1$, for an integer i, are called *regular small factor*.

Let x be a string of length n and let $small(x, i)$ be the small factor of x starting at position $i \leqslant m$. Assume, for technical reasons, that the text has $k-1$ special end-markers at the end. For each i define

$$\underline{x}_i = name(small(x, i)),$$

where $name(f)$ is the name of the regular small factor equal to f (if there is no such factor then it is some special symbol). The names should be consistent:

$$\text{if } small(x, i) = small(x, j) \text{ then } \underline{x}_i = \underline{x}_j.$$

The names are integers from $[1 \ldots M]$. The equality of two names can be checked with constant work. The sequence $\underline{x} = \underline{x}_1 \underline{x}_2 \underline{x}_3 \underline{x}_4 \ldots \underline{x}_m$ is called here the *dictionary of small factors.*

A simple way to compute such dictionary is to use a parallel version of the Karp–Miller–Rosenberg algorithm. This gives $n \cdot \log(\log(n))$ total work, very close to optimal.

The optimality is achieved using a (modified) multipattern matching automata. The following lemma has been proved by Kedem et al.

Lemma 2.3 (Kedem et al. [10]). *The dictionary of small factors can be computed in* $\log n$ *time with linear work.*

The proof given by Kedem et al. is rather complicated. It works for alphabets of arbitrary size. However if the alphabet has a constant size then a rough approach is possible, and the introduction of regular small factors is not necessary. In this case, the lemma above follows by a very simple application of the "four Russian" trick of encoding small segments by numbers as used in [7] and [11] for string problems. We sketch below how it can be done. All small factors receive consistent names (also nonregular ones) which distinguish all nonequal small factors, while in the KLP algorithm nonregular small factors receive a common single name and are undistinguishable.

Assume for simplicity that the alphabet is binary, and let $s = \log(m)/4$. There are potentially only $m^{1/2}$ binary strings of length $2 \cdot s$. For each of them we can precompute names of all small factors of length s; these names could be the integers corresponding to binary representations of factors. The total number of processors is $n/\log(m)$, so we have about $m^{1/2}$ processors for each small segment of logarithmic size. Next, take independently and at the same time each segment of size $2 \cdot s$ starting at a position divisible by s. One processor can encode the segment as a number (in binary representation) in $\log(m)$ time; then we look at the precomputed table for names of its small factors and write down s consecutive entries of string $\underline{x}$ in time $O(s)$.

We introduce in this paper the fifth type of factors: *thin factors.* It is a natural generalization of *small factors* to the 2D case. Thin factors are m by $\log(m)$ subarrays of text or pattern arrays. They arise if we cut the 2D pattern by $m/\log(m)$ *cut-lines*, the

distance between consecutive lines being $\log m$, see Fig. 3. The construction of the dictionary for such factors is discussed in the section on 2D-pattern matching.

We start with the section on one-dimensional string-matching. The algorithm for two dimensions is conceptually a natural extension of the one-dimensional case algorithm.

3. One-dimensional string-matching

The basic parts of one- and two-dimensional pattern-matching algorithms presented in this paper are similar: computation of the dictionary of small factors, compression of strings by encoding disjoint $\log(n)$ sized blocks by their names, and application of the partly optimal matching algorithm (the algorithm of Lemma 2.2). Two auxiliary functions are needed: *shift* and *compress* (*cm*, in short).

Let x be the pattern of size m, y be the text of size n, and let $k=\log m$. We assume that n, m are multiples of k. The pattern and the text are given as vectors of symbols, the first position is 1. For $0 \leqslant r \leqslant k-1$, let us denote (see Fig. 1):

$$shift(x, r)=x[1+r \ldots m-k+r].$$

For a string z whose length is a multiple of h denote:

$$cm(z)=\underline{z}_1 \underline{z}_{k+1} \underline{z}_{2k+1} \cdots \underline{z}_{(h-1)k+1},$$

where $h=|z|/k$. Recall that $\underline{z}_j$ is the name of the small factor (of length k) starting at j. The string $cm(z)$ contains the same information as z, and is shorter by a logarithmic factor. Basically, each letter of the new string $cm(z)$ encodes a logarithmic-size block of z.

Intuitively speaking $cm(z)$ is a concatenation of names of consecutive small factors, which compose a given string. The *compression ratio* is the reduction of information;

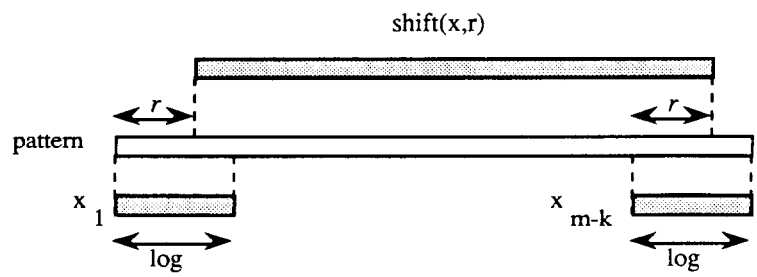

Fig. 1. Illustration of the operation *shift*. To identify the match it is enough to identify the subword $shift(x, r)$ and the first and last full small factors $\underline{x}_1$ and $\underline{x}_{m-k}$ of the pattern x (the three shadowed factors). The identification of $shift(x, r)$ is done by searching for its compressed version in an efficient way due to the fact that it starts at a *regular* position.

in this case a small factor (string of length $\log m$) is replaced by one symbol. The compression ratio is $\log(m)$. One-dimensional objects of size $\log m$ are reduced to zero-dimensional. We later generalize this idea extended for two dimensions in the next section.

A position j in the text is called a *regular position* iff $j \bmod k = 1$. The whole idea is to limit the search to those (starting positions) j which are *regular*. Observe that if j is a regular position then it is the $(j \operatorname{div} k + 1)$th regular position (if we enumerate regular positions by consecutive numbers).

We omit the proof of the following obvious observation:

Fact 3.1. *An occurrence of x starts at position i in the text y iff the following condition is satisfied*

$$\text{COND1}(i)\text{: } cm(shift(x, j-1)) \text{ starts in } cm(y) \text{ at } (j \operatorname{div} k + 1) \text{ and } \underline{x}_1 = \underline{y}_i$$

$$\text{and } \underline{x}_{m-k} = \underline{y}_{i+m-k},$$

where j is the first regular position to the right of i (including i). The value of j can be computed in constant time.

Example. Assume $k = 3$, the length of the pattern x is $m = 15$, and the length of the text y is $n = 31$. The regular positions are $1, 4, 7, 10, \ldots$. Let us write explicitly condition COND1(8). The first regular position to the right of 8 is $j = 10$. Then x occurs at position 8 iff the prefix of x of length 3 occurs at 8, the suffix of x of length 3 (starting at position 13 in x) occurs at position $8 + 15 - 3 = 20$ in y, and the segment $x[3 \ldots 15] = shift(x, 2)$ occurs at a regular position 10 in y. This segment is contained precisely between *cut lines*. These are the vertical lines crossing between positions k and $k+1$, positions $2k$ and $2k+1$, etc. After translating it into the language of compressed objects (where position 10 in y corresponds to position $10 \operatorname{div} 3 + 1 = 4$ in $cm(y)$) our condition is

$$\text{COND1}(8) := [cm(shift(x, 2)) \text{ starts in cm}(y) \text{ at } 4 \text{ and } \underline{x}_1 = \underline{y}_8 \text{ and } \underline{x}_{13} = \underline{y}_{20}].$$

The condition COND1(i) is illustrated by Fig. 2 (see also Fig. 1).

The structure of the algorithm based on the equivalence of a match at a position i and COND1(i) is given below.

Algorithm 1 {one-dimensional pattern matching}
begin
compute dictionary of small factors together for x and y;
find all patterns $cm(shift(x, 0), cm(shift(x, 1), \ldots, cm(shift(x, k-1)$
in the text $cm(y)$ by the partly optimal matching algorithm from Lemma 2.2;
for each position i **do in parallel**
{in constant time by one processor for each i due to information already computed}

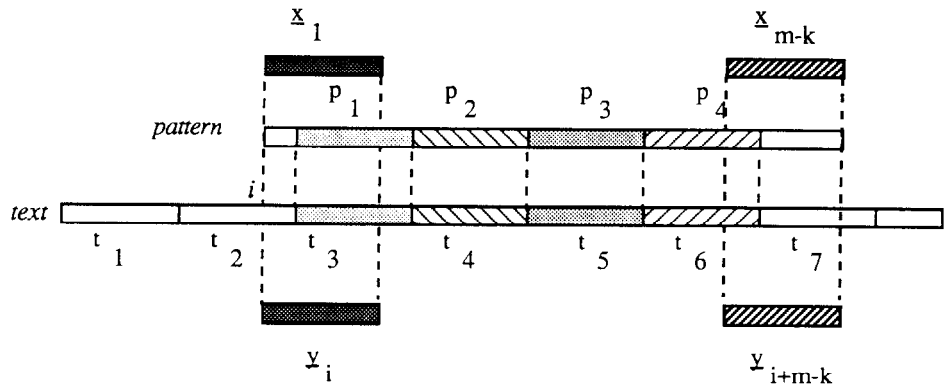

Fig. 2. Let $p_1p_2p_3p_4 = cm(shift(x, k-(i \bmod k)))$ and $cm(y) = t_1t_2t_3t_4t_5t_6t_7$. p_i's are names of logarithmic segments (small factors) of the pattern x and t_i's are names of segments of the text y. The pattern x starts at position i in y iff $p_1p_2p_3p_4$ starts at the 3th position in $t_1t_2t_3t_4t_5t_6t_7$, and $\underline{x}_1 = \underline{y}_i$, and $\underline{x}_{m-k} = \underline{y}_{i+m-k}$.

{**if** COND1(i) is satisfied for i **then** report the match at i
end algorithm.

Theorem 3.2. *Algorithm* 1 *solves the string-matching problem on a CRCW PRAM in* $O(\log(m))$ *time with* $O(n/\log(m))$ processors (*with optimal total work*).

Proof. The compressed text has size $m/\log(m)$, and we have $\log(m)$ shifted compressed patterns, each of size $m/\log(m)$. Hence, according to Lemma 2.2, the total work, when applying the algorithm from this lemma, is of order $m/\log(m) \cdot (m)/\log(m) + \log(m) \cdot m/\log(m)$ which is $O(m)$. This completes the proof. □

4. Recognizing images: 2-dimensional counterpart of Algorithm 1

We give a novel approach with greater utilization of the two-dimensionality of the image. It is based on the fact that the key lemma (Lemma 2.2) works in the same way for strings as for multidimensional images. In [6] the algorithm for 2D patterns is not a multiple application of a 1D-pattern algorithm. It takes fully into account the 2D structure of images.

Let Y be the n by n host array and let X be the m by m pattern array. The general case of rectangular (nonsquare) arrays can be handled in the same way. However we should assume that the shorter side is at least $\log m$ long, that is, the pattern is not thin. Otherwise Algorithm 1 can be differently used. We say that a subarray occurs at position (i, j) in a given table iff when its left upper corner is placed at position (i, j) it coincides with the aligned part of the table.

There are possible at least two algorithmic approaches to the 2D-pattern matching. The first one is to parallelize Baker's and Bird's algorithms. Such parallelization uses the multi-text/multi-pattern approach. The two-dimensional matching is reduced to a one-dimensional matching by multi-text/multi-pattern matching. The second approach is to consider Algorithm 1. This means that Algorithm 1 should have an easy extension to multi-dimensional case. We show that this is true indeed.

The cut-lines are the columns $\log(m), 2\cdot\log(m), \ldots, n-\log(m)$ of the text array, see Fig. 3. There are $n/\log(m)$ cut lines.

We cut two-dimensional patterns by cut-lines in the same way as strings are cut. These lines cut the pattern into small 2D pieces: thin factors. Similarly as in Algorithm 1 our idea is now to cut the pattern image into small pieces and make compression, this time the pieces are two dimensional.

We assume for simplicity of presentation that we deal with pattern images whose sides have lengths divisible by $k=\log(m)$.

For an array Z of shape n' by n' denote by $Z_{[j]}$ the jth column of Z. The jth thin factor of Z is the rectangle of shape n' by k whose first column is $Z_{[j]}$. Denote by $\underline{Z}_{[j]}$ the compressed version of the jth thin factor, this is the vector of size n', whose ith component is the compressed name of the ith row of the thin factor:

$$\underline{Z}_{[j]}(i)=\text{name of } Z(i,j),\ Z(i,j+1),\ldots,Z(i,j+k-1).$$

The set of vectors $\underline{Z}_{[j]}$ for $1\leqslant j\leqslant n'-k$ is called the dictionary of thin factors. This dictionary can be obtained by computing the dictionary of small factors (in a one-dimensional setting) for all rows of the array Z independently. We assume later that it has been computed.

Let X be the pattern image of length m, Y be the text of length n and $k=\log m$. We assume that n, m are multiples of k. Denote by $X_{[j]}$ the jth column of X. For $0\leqslant r\leqslant k-1$ denote

$$\text{SHIFT}(X,r) \text{ is the rectangle composed of columns } X_{[1+r]},\ldots,X_{[m-k+r]}.$$

For a rectangle Z whose width is a multiple of k denote:

$$\text{CM}(Z)=\underline{Z}[1]\underline{Z}[k+1]\underline{Z}[2k+1]\ldots\underline{Z}[(h-1)k+1],$$

where $h=|z|/k$. The rectangle CM(Z) contains the same information as z and is shorter by a logarithmic factor. Essentially each column of the new rectangle encodes a thin factor. The column $\underline{Z}_{[jk+1]}$ is the compressed version of columns $jk+1$, $jk+2,\ldots,jk+k-1$ of the original array Z. Intutively speaking, CM(Z) is a concatenation of compressed consecutive thin factors, which compose a given pattern image. Each thin factor is reduced to a single column. The *compression ratio* is in this case $\log(m)$. Two-dimensional objects (thin factors) are reduced to one-dimensional (single columns). We use the same idea as in the former section. We say that a column j' is regular iff $j' \bmod k=1$. We use in the algorithm the following obvious fact (see also Fig. 3).

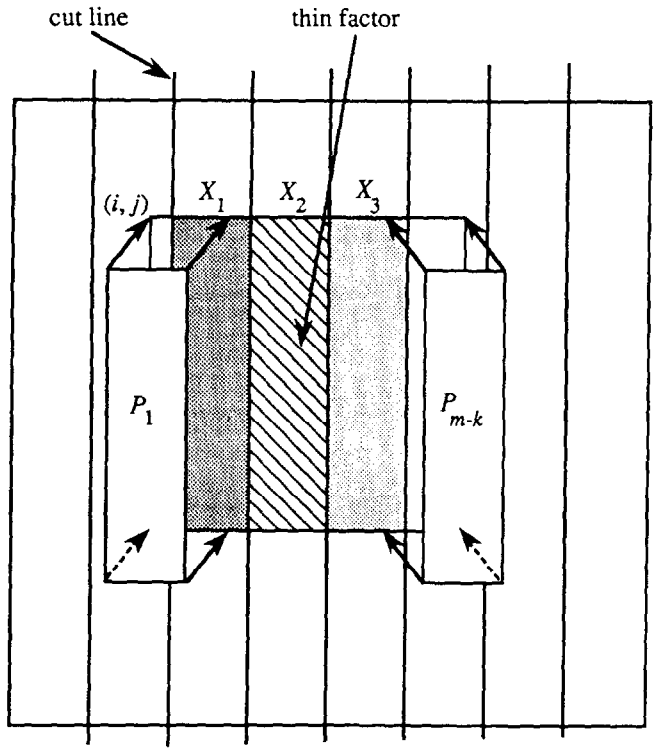

Fig. 3. Partitioning of the pattern and the text arrays by cut-lines. We search for the "essential" part of pattern image (composed of dashed thin factors), and the first and last thin factors of X.

Fact 4.1. *An occurrence of X occurs at (i, j) in the image Y iff the following conditions are satisfied*

COND2(i, j):

CM(SHIFT$(X, j'-j)$ *occurs in* CM(Y) *at position* $(i, j' \operatorname{div} k+1)$;

$\underline{X}_{[1]}$ *occurs at* ith *place of* jth *column of* $\underline{Y}$;

$\underline{X}_{m-k}$ *occurs at* ith *place of* $(j+m-k)$*th column of* $\underline{Y}$,

where j' is the number of the first regular column to the right of j (including j).

The structure of the algorithm given below is essentially the same as that of Algorithm 1.

Algorithm 2 {two-dimensional pattern matching}
begin
compute dictionary of small factors together for rows of X and Y; construct tables $\underline{X}$ and $\underline{Y}$;
find patterns CM(SHIFT(X, 0), CM(SHIFT(X, 1), ..., CM(SHIFT(X, $k-1$)
in the image CM(Y) by the algorithm from Lemma 2.2;
for each j **do in parallel**
find all occurrences of the first and last column of $\underline{X}$ in the jth column of $\underline{Y}$ {of the first and last thin factor of image pattern, one-text/one-pattern algorithm}
for each position (i, j) of Y **do in parallel**
{in constant time by one processor for each (i, j) due to information already computed}
if conditions COND2(i, j) are satisfied **then** report the match at (i, j)
end algorithm.

Theorem 4.2. *Under the CRCW PRAM model, Algorithm* 2 *is an optimal parallel* $\log(n)$*-time algorithm for two-dimensional pattern matching.*

Proof. The proof is essentially the same as that of Theorem 3.2. In this case the compressed image has size $M/\log(M)$ and we have $\log(m)$ shifted compressed patterns, each of size $m/\log(m)$. A two-dimensional version of Lemma 2.2 applies. This completes the proof. □

5. Concluding remark

The Kedem–Landau–Palem algorithm is a rough algorithm. It does not use any special mathematics on strings or images. Nevertheless, it is very efficient and easy to understand. The idea of cutting an object into small pieces and compressing them is a useful one. In this paper, we introduced a new notion of thin factors. We consider that the notion of small two-dimensional pieces of the image (thin factors) is a natural notion when dealing with two-dimensional objects, and could be helpful in other applications. Our approach to two-dimensional pattern matching is here more oriented to the two-dimensionality of the problem. It also shows applicability of cutting/compression technique.

Acknowledgment

We are grateful to Gadi Landau for pointing out nontriviality of the 2D-pattern matching for constant size alphabets, and activating our work on optimal parallel computation for the problem.

References

[1] A. Amir and G. Landau, Fast parallel and serial multidimensional approximate pattern matching, *Theoret. Comput. Sci.* **81** (1991) 97–115.

[2] A. Apostolico, C. Ilipoulos, G. Landau, B. Schieber and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica* **3** (1988) 347–365.

[3] T. Baker, A technique for extending rapid exact string matching to arrays of more than one dimension, *SIAM J. Comput.* **7** (1978) 533–541.

[4] R.S. Bird, Two-dimensional pattern-matching, *Inform. Process. Lett.* **6** (1977) 168–170.

[5] M. Crochemore and W. Rytter, Parallel computations on strings and arrays, in: Choffrut and Lengauer eds, *(STACS'90)*, Lecture Notes in Computer Science **415** (Springer, Berlin, 1990) 109–125.

[6] M. Crochemore and W. Rytter, Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays, *Theoret. Comput. Sci.* **88** (1991) 59–82.

[7] Z. Galil, Optimal parallel algorithm for string-matching, *Inform. and Control* **67** (1985) 144–157.

[8] A. Gibbons and W. Rytter, *Efficient parallel algorithms* (Cambridge University Press, Cambridge, 1988).

[9] R. Karp, R. Miller and A. Rosenberg, Rapid identification of repeated patterns in strings, arrays and trees, in: *Proc. 4th ACM Symp. on Theory Of Computation* (1972) 125–136.

[10] Z. Kedem, G. Landau and K. Palem, Optimal parallel suffix–prefix matching algorithm and its applications, *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures* (1989) 388–398.

[11] M.G. Main and R.J. Lorentz, Linear-time recognition of square-free strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, NATO Advanced Science Institutes, Ser. F, Vol. 12 (Springer, Berlin, 1985) 271–278.

[12] U. Vishkin, Optimal parallel pattern matching in strings, *Inform. and Control* **67** (1985) 91–113.